Parallel implementation of a spatio-temporal visual saliency model

Anis Rahman, Dominique Houzet, Denis Pellerin, Sophie Marat, Nathalie Guyader

Noname manuscript No.
(will be inserted by the editor)

Parallel Implementation of a Spatio-temporal Visual Saliency Model

A. Rahman · D. Houzet · D. Pellerin · S. Marat · N. Guyader

Received: date / Accepted: date

Abstract The human vision has been studied deeply in the past years, and several different models have been proposed to simulate it on computer. Some of these models concerns visual saliency which is potentially very interesting in a lot of applications like robotics, image analysis, compression, video indexing. Unfortunately they are compute intensive with tight real-time requirements. Among all the existing models, we have chosen a spatio-temporal one combining static and dynamic information. We propose in this paper a very efficient implementation of this model with multi-GPU reaching real-time. We present the algorithms of the model as well as several parallel optimizations on GPU with higher precision and execution time results. The real-time execution of this multi-path model on multi-GPU makes it a powerful tool to facilitate many vision related applications.

Keywords visual saliency · spatio-temporal model · parallel implementation · graphics processors

1 Introduction

Visual attention models mimic the capacity of a primate's visual system to focus on particular places in a visual scene. These models tend to reduce the spotlight of focus to a single object or a portion of the visual scene called the salient regions that guide the attention by locating the spatial discontinuities using different channels like intensity, color, orientation, motion, and many others.

The bottom-up spatio-temporal visual saliency model [14] discussed here is inspired from the primate's visual system, and is modeled all the way from the retina to visual cortex cells. This visual saliency model is used to determine where the source of attention lies and the amount of concentration used to contribute or initiate other tasks. This model is interesting because: firstly, the model is linearly modeled all the way from the retina to cortical cells. Secondly, the retinal output causes the separation of useful information into two distinct signals that are more efficient to process. Thirdly, motion compensation used in dynamic pathway that extracts only the moving parts against its background, and motion estimation is used to carry out the motion contrast map. Lastly, the saliency outputs from both static and dynamic pathways are fused together to get the final saliency map. This fusion is done using several adaptive coefficients like maximum and skewness. All these points contribute a step to mimic the human visual system. The model has been analyzed against a large number of images, and then the produced results have been compared against the behavior of human visual system. As an experiment, an eye tracker has been used to evaluate the model as a good predictor of eye movements, and to demonstrate the efficiency of the model. The resulting saliency map can be used to predict such areas, finding its applications in robotics, video content analysis, video reframing process to deliver comforting viewing experience on mobile devices, in video compression, video synthesis.

The motivation behind designing biologically-inspired models is to build robust and versatile vision systems that can adapt to various environmental conditions, users, and tasks.

A. Rahman · D. Houzet · D. Pellerin · S. Marat · N. Guyader
GIPSA-lab
Grenoble, France.
E-mail: anis.rahman@gipsa-lab.grenoble-inp.fr

D. Houzet
E-mail: dominique.houzet@gipsa-lab.grenoble-inp.fr

D. Pellerin
E-mail: denis.pellerin@gipsa-lab.grenoble-inp.fr

S. Marat
E-mail: sophie.marat@gipsa-lab.grenoble-inp.fr

N. Guyader
E-mail: nathalie.guyader@gipsa-lab.grenoble-inp.fr

Mostly, visual saliency models involve many computationally intensive tasks, making its implementation in real-time environments on a single processor impossible. This limiting factor also restricts the inclusion of other complex processes into the existing model. Hence, real-time solution is achievable only by the simplification of the entire pathway, as demonstrated by Itti [8] and Nabil et al. [16]. Over the years, computer graphics hardware has evolved into completely programmable shader architecture from fixed function architecture. Together with a programming model like CUDA [1] makes it a desirable choice to leverage the computational power of graphics hardware for general-purpose computations. These devices are also cheap, accessible to everyone, and easier to program. Thus, graphics devices may be a suitable platform to accelerate many visual attention algorithms.

This model presented above [14] mimics human visual perception from retina to cortex using both static and dynamic information and hence compute-intensive. In this article, we propose parallel adaptation of this visual saliency model onto GPU. After this transformation, we apply several optimizations to leverage the raw computational power of graphics hardware. Subsequently, proposing a real-time solution on multi-GPU, and demonstrating the performance gains. In the end, we also evaluate the effects of lower precision on the resulting saliency map of our model.

The article is organized as follows: in section 2, we present a brief overview of the prior work for accelerating visual saliency models using parallel platforms. In section 3, the main steps of the visual saliency model implemented are described. In section 4, the architecture of NVIDIA graphics cards is presented, and its programming model is detailed. In section 5, we describe our GPU implementation, and different optimizations to improve speedups. Section 6 reports the achieved speedups, and also validity of these results is evaluated. In the end, conclusion of the article and its future prospects are discussed.

2 Related Work

In the past few decades, different approaches have been developed to model human visual perception, which find their application in computer vision systems. One of the famous saliency model proposed by Itti and Koch [9] decomposes the visual input into multiple feature maps, which are afterwards combined to form a saliency map. Depending on the application, various improvements and optimizations are made into the existing model that makes it more compute-intensive with larger execution times. This limiting factor restricts its use for real-time systems like robotic vision. As a result, a few attempts of parallelization [18,5] are made by using high performance systems like cluster of computers to achieve real-time capability, but these systems are complex to develop and manage.

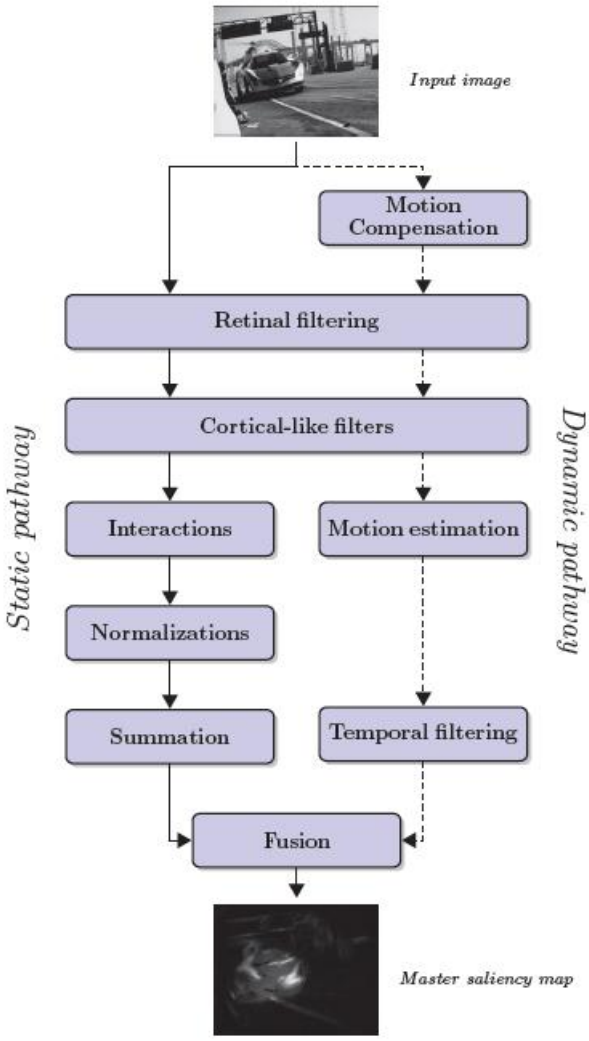


Fig. 1 The spatio-temporal visual saliency model

With the introduction of inexpensive graphics devices with enormous computational power, more attempts are made to parallelize the visual saliency models, for example Longhurst et al. [12] used the saliency map for selective rendering; Mantiuk et al. [13] combined real-time rendering, MPEG4 video compression and visual attention to deliver real-time low-bandwidth computer animation. All these parallel implementations use channels for the static information only, and also use simplified versions of the static model to get real-time capability. Peters [17] used a visual attention for animating virtual characters. The saliency model is partially implemented on the GPU, as feature map calculation time for color and intensity is faster on CPU than on GPU. Recently, CUDA programming model for general-purpose computations on the graphics hardware is introduced by NVIDIA; making the programming completely independent of graphics pipeline. Implementations using the newer programming

model by Xu et al. [20] implements visual attention for static information only. A complete parallel implementation of the model including the static and dynamic modalities using CUDA by Lee et al. [10] for tracking of visually attended objects in the virtual environment resulted in real-time processing for image sizes of 256×256 pixels. But, this implementation does not use real visual scenes that make the calculation of feature maps complex.

3 Visual Saliency Model

The bottom-up model [14] illustrated in figure 1, is inspired from the primate's visual system. This model is sub-divided into two distinct pathways: static and dynamic pathways.

3.1 Static pathway

Retina model is primarily based on the primate's retina, which imitates the photoreceptor, horizontal, bipolar, and ganglion cells. To begin with, the photoreceptor cells carry out luminance extraction by removing high frequency noise using a low-pass retinal gaussian filter. Subsequently, the output of photoreceptor cells is passed on as input to the horizontal cells; also a function of low-pass filter. The response from these cells is twice than the previous retinal low-pass filter. Down the line are the bipolar cells acting as a high-pass filter, which simply calculates the difference between outputs 'y' and 'h' from photoreceptor and horizontal cells respectively. The bipolar output can be designed to consist of two modes: if 'ON' than positive part of the difference is kept, otherwise the absolute value when 'OFF'.

$$p = ON - OFF$$
$$where\ ,\ on = |y-h|$$
$$off = |h-y|$$

The model produces two types of outputs: the parvocellular output that enforces equalization of the visual by increasing its contrast, consequently, increasing the luminance of low intensity parts in the visual. Next in the order, the magnocellular output responds to higher temporal and lower spatial frequencies. Analogous to primate's retina, the ganglion cells respond to high contrast and the parvocellular output highlights the borders among the homogeneous regions, thus exposing more detail in the visual.

Cortical-like filters is a model of simple cell receptive fields that are sensitive to visual signal orientations and spatial frequencies. This can be imitated using a bank of gabor filters organized in two dimensions, that is closely related to the processes in the primary visual cortex. A Gabor function is defined as:

$$G(u,v) = exp\left\{-\left(\frac{(u\prime - f_0)^2}{2\theta_u^2} + \frac{v\prime^2}{2\theta_v^2}\right)\right\}$$
$$where,\ u\prime = u\cos\theta + v\sin\theta$$
$$v\prime = v\cos\theta - u\sin\theta$$

The retinal output is filtered using gabor filters implemented in frequency domain, after applying a mask. The mask is similar to a Hanning function to produce non-uniform illumination approaching zero at the edges. The visual information is processed in different frequencies and orientations in the primary cortex i.e. the model use 6 orientations and 4 frequencies to obtain 24 partial maps. These filters demonstrate optimal localization properties and good compromise of resolution between frequency and spatial domains.

Interactions In primate visual system, the response of cell is dependent on its neuronal environment; its lateral connections. Therefore, this activity can be modeled as linear combination of simple cells interacting with its neighbors. This interaction may be inhibitory or excitory depending on the orientation or the frequency: excitory when in the same direction, otherwise inhibitory.

$$E_{int}(f_i,\theta_j) = E(f_i,\theta_j).w$$
$$where,\ w = \begin{cases} 0.0 & -0.5 & 0.0 \\ 0.5 & 1.0 & 0.5 \\ 0.0 & -0.5 & 0.0 \end{cases}$$

The produced maps are the image's energy in function of the spatial frequency and orientation; after taking into account the interactions among different orientation maps.

Normalizations The intermediate energy maps from the visual cortical filters and interaction phase are normalized. This model uses a technique proposed by Itti et al. [9] for strengthening the intermediate results.

Summation Ultimately, a saliency map for the static pathway is extracted for the input visual, simply by summing up all the energy maps. It is significant that the resulting map has salient regions; one's with highest energy, which can be observed in the figure 3(d) by energy located on objects appearing to be salient.

$$E_{salient} = |E_{int}(f_i,\theta_j)|$$

3.2 Dynamic pathway

On the other hand, the dynamic pathway finds salient regions from a moving scene.

is the compensation of the background motion to estimate the relative motion of regions against background

Pre-processing The dynamic pathway performs camera motion compensation [15] of regions with relative motion against their background. This compensation is immediately followed by retinal filtering to illuminate the frame before passing it to the next stage of the pathway.

Motion estimation [3] is used to find local motion with respect to the background. The algorithm is based on gabor filters to decompose the image into its sub-bands. These equations are then used to estimate the optical flow between two images. We use a bank of N gabor filters, with the same radial frequency and the same spatial support σ, we can then adjust the parameter θ as a function of ith sub-band as $\theta = i\pi/N$. After gabor filtering, we calculate a system of N equations for each pixel at each time t using spatial and temporal gradients to get an oversized system as following:

$$\begin{pmatrix} \Omega_2^x & \Omega_1^y \\ \Omega_2^x & \Omega_2^y \\ \cdots & \\ \Omega_n^x & \Omega_n^y \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} \Omega_2^t \\ \Omega_2^t \\ \cdots \\ \Omega_n^t \end{pmatrix}$$

To resolve this oversized system, which is fairly noisy having some errors that are relatively low, but others completely absurd making the entire system unstable. To minimize these squared residuals, we use the method of iterated weighted least squares within the motion estimator.

Temporal filtering is the process of modifying the sequence of images based on its temporal information. Usually, such filtering is used to remove excessive noise and extraneous information. In the model, the motion vectors are calculated using modalities like speed, orientation, and direction. To remove noise from these motion vectors, we use temporal median filtering between current image's motion vector and its 4 predecessors to eliminate the noise added. Finally, we get the dynamic saliency map.

3.3 Fusion

The saliency maps from both the static and dynamic pathways exhibit different characteristics i.e. static saliency map has larger salient regions based on textures, whereas dynamic saliency map has smaller salient regions depending on the moving objects. Based on these features the two saliency maps M_s and M_d from static and dynamic pathways are fused together using:

$$Saliency\ map = \alpha M_s + \beta M_d + \gamma(M_s \times M_d)$$
$$where, \begin{cases} \alpha = max(M_s) \\ \beta = skewness(M_d) \\ \gamma = max(M_s) \cdot skewness(M_d) \end{cases}$$

where static and dynamic maps are modulated using maximum and skewness respectively. On the other hand, the reinforcement parameter γ is used to include the regions that have low motion, but has large salient regions in static saliency map. In the end, we get a final saliency map for the attention model.

4 NVIDIA GPUs

The newer graphics cards like NVIDIA's Geforce GTX 480 [11] implement massively parallel architecture, comprising of 480 scalar processors (SPs) running at 1.35 GHz each. It achieves the maximum utilization of the hardware computing units by launching and executing massive number of threads. The graphics hardware comprises of numerous stream processors that when grouped together provide huge computing power doing parallel processing. A single instruction is executed across all the processors in the group that are associated to specialized hardware for texture filtering, texture addressing, cache units and fast on-chip shared memory. All these grouped processors can communicate using the shared memory space. The new design delivers impressive computational power, which is made possible by the management of numerous threads on fly along with high memory bandwidth.

A major breakthrough was the introduction of BrookGPU [4], that is, a compiler for stream programming language an extension to C that hid the graphics API. It facilitated the parallel programmers to use the GPU as a co-processor. This attempt steered the market towards GPU-assisted parallel computing bypassing the need for graphics APIs. This new model is not linear like the traditional pipeline model, but here the data circulates during its processing. The same team developing BrookGPU at NVIDIA came up with CUDA (Compute unified device architecture) [1]. CUDA provides a platform that is more suitable and efficient for GPGPU computing. The language used is an extension to familiar C; making the learning curve easier. The process of transformation of algorithm is further eased by the use of GPU-specialized libraries. On the whole, all these contribute towards the maximum utilization of powerful execution units, as well as, alleviation of memory wall problem.

Recently, a new open cross-platform standard OpenCL [7] is brought to light to get more out of multicore processors including GPUs. It not only allows to take advantage of task-level parallelism and data-level parallelism, but also allows the flexibility to support CPU-optimized as well as GPU-optimized code. Hence, it is a giant stride towards general-purpose parallel programming of heterogeneous systems.

5 GPU Implementation

The code is composed of host (CPU) and kernel (GPU) code. The host code is responsible for transferring the data to and from the GPU's global memory and afterwards initiates the kernel code through a function call. The kernel code is compiled by the nvcc compiler supplied by NVIDIA. The structure of parallel code for every single thread is clear and flexible. On the whole, the threading model exploits fine-grain data and thread parallelism across the threads nested within coarse-grain data and task parallelism across the thread blocks. This granularity makes the compiled CUDA code scalable, executable on large number of processors.

The only way to achieve high performance is to exploit the multi-core architecture using parallelism. The CPUs provide task-level parallelism, whereas GPUs implement data-parallelism. This makes many computer vision algorithms well-suited to port onto GPUs that are intrinsically data parallel, and require interactivity. The previous graphics-centric programming environments made this porting quite complex, whereas newer CUDA programming model hides all the details; the programmer no longer need to worry about the pipeline, pixels, or textures.

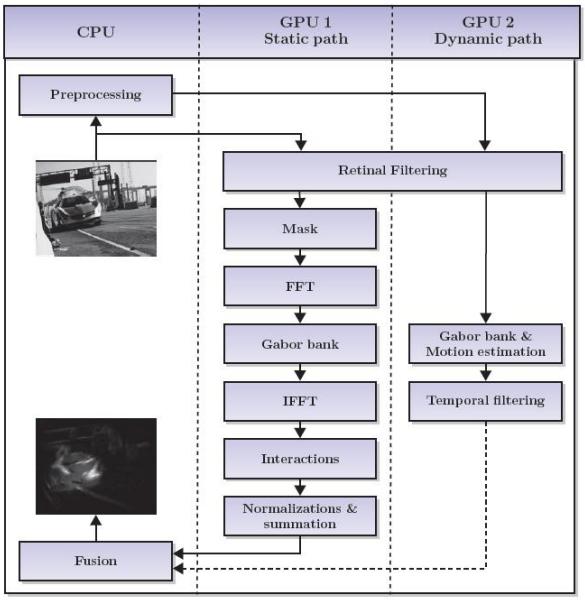


Fig. 2 GPU implementation of the visual saliency model

5.1 The static pathway

To start with the mapping of the algorithm 1 onto GPU, it is partitioned into data-parallel portions of code that are isolated into separate kernels. Then, the input data is transferred and stored on the device memory, which is afterwards used by the synchronous kernels. After all the memory declarations on device, the host sequentially initiates all the data-parallel kernels. First, some preprocessing using retinal filter and hanning mask is done to give more detail to the visual input. Second, the visual data in frequency domain is treated with a 2D gabor filter bank using 6 orientations and 4 frequency bands; resulting in 24 partial maps. Third, the pathway is moved back to spatial domain before doing the interactions among the different partial maps. These interactions inhibit or excite the data values depending on the orientation and frequency band of a partial map. Fourth, the resulting values are normalized between a dynamic range before applying Itti's method for normalization, and suppressing the values lower than the threshold. Finally, all the partial maps are accumulated into a single map that is the saliency map of static pathway.

input : An image Im of size $w \times l$
output: A saliency map

```
1  map ← RetinalFilter(Im);
2  map ← FFT(map);
3  for i ← 1 to orientations do
4      for j ← 1 to frequencies do
5          maps[i, j] ← GaborFilter(map, i, j);
6          maps[i, j] ← IFFT(maps[i, j]);
7          maps[i, j] ← Interactions(maps[i, j]);
8          maps[i, j] ← Normalizations(maps[i, j]);
9      end
10 end
11 saliency ← Fusion (maps);
```

Algorithm 1: Static pathway of visual saliency model

5.2 The dynamic pathway

Similar to the implementation of static pathway, we first perform task distribution of the algorithm and realize a sequential version. Some of the functional units are: recursive gaussian filter, gabor filter bank to break image into subbands of different orientations, biweight tukey motion estimator, Gaussian prefiltering for pyramids, spatial and temporal gradient maps for estimation, and bilinear interpolation. After testing these functional units separately, they are put together to give a complete sequential code. The algorithm being intrinsically parallel allows it to be easily ported to CUDA parallel code.

The algorithm 2 describes the dynamic pathway, where first camera motion compensation and retinal filtering is done as a preprocessing on the visual input. Afterwards, the preprocessed input is passed onto the motion estimator implemented using 3rd order gabor filter banks. The resulting mo-

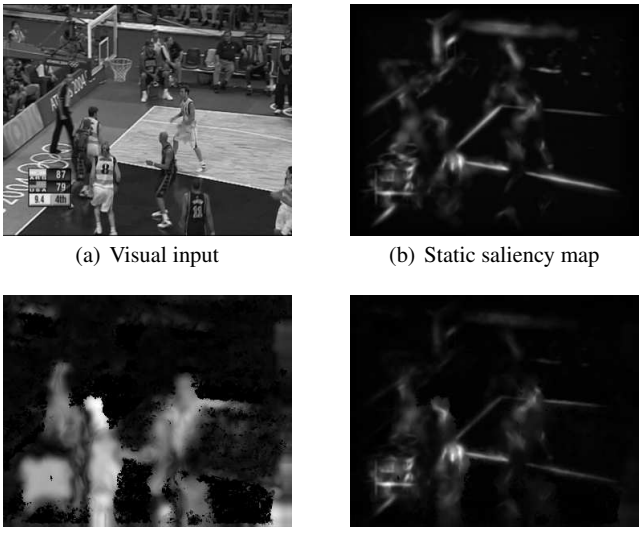
tion vector are normalized using temporal information at the end to get a dynamic saliency map.

input : An image Im of size $w \times l$
output: A dynamic saliency map

```
1 map ← MotionCompensation(Im);
2 map ← RetinalFilter(map);
3 map ← MotionEstimation(map);
4 saliency ← TemporalFilter(map);
```

Algorithm 2: Dynamic pathway of visual saliency model

The saliency maps from both the static and dynamic pathways are copied back onto the host CPU, where they are fused together outputting a saliency map. The two saliency maps from different pathways, and the final output saliency map is shown in the figure 3.



(a) Visual input (b) Static saliency map

(c) Dynamic saliency map (d) Master saliency map

Fig. 3 Results of the visual saliency model

5.3 Memory optimizations

One of the biggest challenges in optimizing GPU code for data dominated applications is the management of the memory accesses, which is a key performance bottleneck. Memory latency can be several hundreds even thousands of clock cycles. This can be improved first by memory coalescing when memory accesses of different threads are consecutive in memory addresses. This allows the external memory controllers to execute burst memory accesses. The second optimization is to avoid external memory accesses through the use of a cache memory, internal shared memories, or registers. The knowledge of the memory access patterns is fundamental to optimize and reduce memory accesses through prefetching and overlapping of memory transfers with computation. This overlapping is naturally performed by the multithreading mechanism of today's GPU. With newer hardware and drivers the memory access model is less restrained, and in future easier programming will be possible without worrying about memory coalescing.

input : Data values from Gabor filter bank in complex format
output: Converted data values to be used for interactions

```
1  x ← blockIdx.x × blockDim.x + threadIdx.x/2;
2  y ← blockIdx.y × blockDim.y + threadIdx.y;
3  mod ← threadIdx.x % 2;
4  pt ← threadIdx.x/2 + 40 × mod;
5  __shared__ float maps_smem[orientations ×
   frequencies][32], buf[72];
6  for i ← 1 to orientations × frequencies do
7      buf[pt      ] ← maps[i][y × w + x      ][mod];
8      buf[pt + 16] ← maps[i][y × w + x + 16][mod];
9      maps_smem[i][threadIdx.x] ← abs(
       buf[threadIdx.x      ] × buf[threadIdx.x      ] +
       buf[threadIdx.x + 40] × buf[threadIdx.x + 40])
       /(w × l);
10 end
11 Interactions(maps_smem);
```

Algorithm 3: The interactions kernel

5.3.1 Coalesced global memory accesses

One of the global memory optimization is coalesced memory accesses to increase the memory bandwidth, and to minimize the bus transactions. Coalescing means that adjacent threads cooperate to load a contiguous segment of global memory in a single read operation. The best case is when one bus transaction is issued for all the threads in a half-wrap. The requirement is that all the threads in the wrap access memory in a sequence i.e. kth word in global memory is accessed by kth thread in the wrap. Thus, the starting address and alignment is important.

5.3.2 Shared memory

Shared memory is an on-chip high bandwidth memory shared among all the threads on a single SM. It provides high performance and communication among the threads of a thread block. Such memory can be implemented effectively in hardware translating to faster memory accesses. Here are different shared memory usages:

– registers extension, to avoid swapping of registers to global memory

- stack, for sub-programs calls and parameters
- arrays, for intermediate results to avoid global memory accesses
- fast communications between threads (on locks, arrays, ...)
- preload of coalesced global memory data followed by uncoalesced shared memory access
- prefetch of data in shared memory used as a data cache managed by software to benefit from spatial and temporal locality of data

Coalescing A method to avoid non-coalesced memory accesses is by re-ordering the data in shared memory. To demonstrate the uses of shared memory, we take an example kernel as shown in algorithm 3, and is illustrated using block diagram shown in figure 4. Here, the data values in complex format consisting of two floats for real and imaginary parts. The very first step is fetching the values into the shared memory, where each float is read by a separate thread i.e. two threads for every complex number as shown in line 3. These global memory accesses are coalesced as contiguous floats are read. Furthermore, we use two shared buffers; one for real part, and the other for imaginary part in line 7, 8. This arrangement gives coalesced shared memory accesses during computation in line 9 to convert the complex numbers into real, and also to scale down the output from the unnormalized Fourier transforms done using CUFFT library.

Bank conflicts Shared memory is similar to a local scratchpad that is 16 KB with 16×1 KB banks, and these banks can service only one address at a time. There are two cases when there is no bank conflict:

- If all the threads of a half-wrap access different banks
- If all the threads of a half-wrap access the same address

In case if multiple threads access the same bank causes conflicts. These conflicting accesses are required to be serialized either using an explicit stride based on thread's ID or by allocating more shared memory. In our case, when thread's ID is used to access shared memory then a conflict occurs, as thread 0 and 1 access the same bank. Thus, we use a stride of 8 to avoid any conflicts as shown in the line 4. Although, a multiprocessor takes only 4 clock cycles doing a shared memory transaction for the entire half wrap, but bank conflicts in the shared memory can degrade the overall performance.

Prefetching Another use of shared memory is to prefetch data from the global memory, and cache it in shared memory. In the example kernel, the data after the conversion and rescaling is cached, and this prefetched data is used for the next phase of applying the interactions, as shown in line 11of the example kernel.

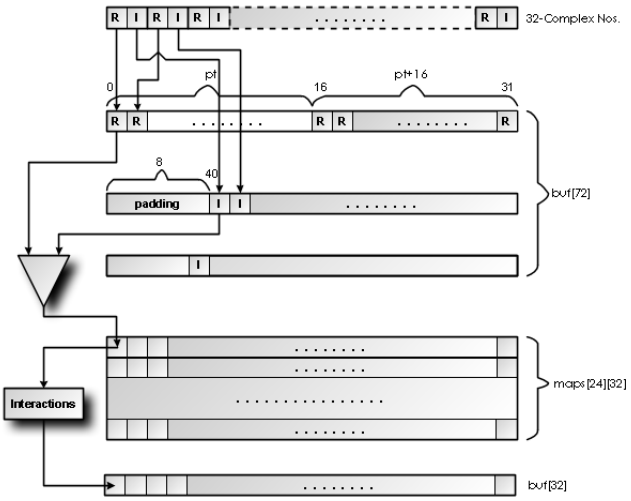


Fig. 4 Block diagram of data-parallel interaction kernel

Reducing device memory accesses In MCPI_Prefetch kernel, we use spatial and temporal gradient values to get $N(2N-1)$ solutions that are used to perform the iterative weighted least square estimates. These numerous intermediate values are stored as arrays variables because the register count is already high. Unfortunately, this leads to costly global memory accesses that can be avoided by placing some values in shared memory. Consequently, we get a solution with less number of global memory access, and efficient use of limited resources on the device. We achieved performance gains by carefully selecting the amount of shared memory without compromising the optimal number of active block residing on each SM.

Reducing register count In MCPI kernel there is a limitation of higher register count due to the complexity of the algorithm, hence, resulting in reduced number of active thread blocks per SM. In our nave solution, the register count is 22 that can be considerably reduced reduced to 15 registers per block using shared memory for some local variables. Consequently, the occupancy increased from .33 to .67 with optimal number of thread block residing on each SM. These variables to be placed in shared memory are carefully selected to reduce the number of synchronization barriers needed.

5.3.3 Texture memory

Texture memory provides an alternative path to device memory, which is faster. This is because of specialized on-chip texture units with internal memory to allow buffering of data from device memory. It can be very useful to reduce the penalty incurred for nearly coalesced accesses. In our implementation, the main motion estimationMCPI kernel exhibits

a pattern that requires the data to be loaded from device memory multiple times. This leads to performance degradation because of high device memory latency. As a solution, we employed texture memory's caching mechanism to prefetch data to reduce global memory latency, hence leading to 10% performance improvement of the problematic kernel.

5.4 Real-time streaming solution

OpenVIDIA [6] and GpuCV [2] are open source libraries that provide an interface for video input, display and programming on GPU using a bunch of high-level implementations of various image processing and computer vision algorithms. Some example implementations include feature detection and tracking, skin tone tracking, projective panoramas, and many more.
After the parallel implementation of the visual saliency algorithm, we used OpenVIDIA to demonstrate the real-time processing. The demonstration is done on a single core machine with a graphics card installed, and the library is used to interface with the webcam. This resulted in execution of the visual saliency model with frame size of 320×240 pixels at 22 fps on a 2-GPU GTX 285 shared device platform as shown in figure 5. Hence, making evident the use of GPUs for real-time processing.

Fig. 5 Platform for real-time solution

6 Results

All implementations are tested on a 2.67 GHz quad-core system with 10GB of main memory, and Windows 7 running on it. On the other hand, the parallel version is implemented using latest CUDA v3.0 programming environment on NVIDIA Geforce GTX 480. The static pathway is evaluated with image sizes of 640×480 and 512×512 pixels, whereas the dynamic pathway uses several datasets of image sequences with sizes ranging from 150×150 to 316×252 pixels.

6.1 Speedup of static pathway

In the algorithm, a saliency map is produced at the end of static pathway, which identifies the salient regions in the visual input. These stages include a Hanning mask, Retinal filter, Gabor filter bank, interaction, normalization and fusion. All these stages show a great potential to be parallelized, and are isolated within separate kernels. Initially, the entire pathway is implemented using MATLAB that happens to be extremely slow because it involves a number of compute-intensive operations; for example: 2D-convolutions, conversions between frequency and spatial domains, and Gabor banks producing 24 partial maps that are further processed. As a result, a single image would take about 18s to pass through the entire pathway; making it unfeasible for real-time applications.
The target of the second implementation in C is to identify

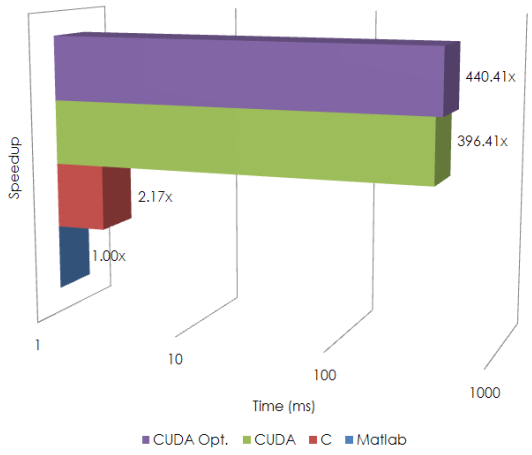


Fig. 6 Speedups for static pathway of the model

the data-parallel portions after writing it in a familiar language. It includes many optimizations and also the use of highly optimized FFTW library for Fourier transforms, but the speedup witnessed is only 2.17x.
At first, the porting of the data-parallel portions into separate kernels for the GPU can be simple. But, the code requires many tweaks to achieve the promised speedup, which happens to be the most complex maneuver. Although, the very first implementation involves partitioning into data-parallel portions, which results in speedup about 396x, as shown in the figure 6. The peak performance topped over to 440x af-

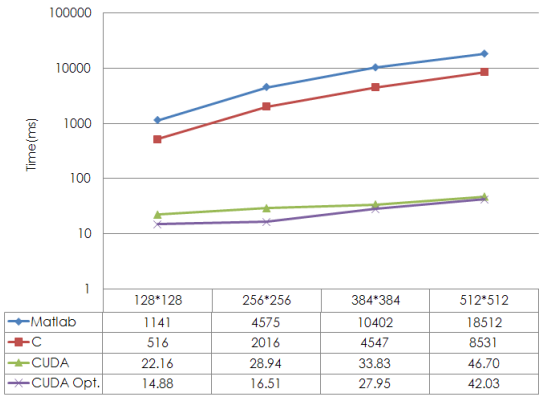


Fig. 7 Timings for different image sizes

Table 1 Timings for the dynamic pathway after optimizations

	treetran	treediv	yosemite
Matlab	13.3s	12.86s	46.61s
C	1.75s	1.76s	6.28s
CUDA	54ms	54ms	76ms

ter making various optimizations on GTX 480. Also, the figure 7 shows the timings for the different implementations.

6.2 Speedup of dynamic pathway

To evaluate the performance gains of the dynamic pathway, we compare the timings on GTX 480 device against the sequential C and MATLAB code as shown in table 1. We used three datasets of images treetran, treediv and yosemite for comparison, the first two with resolution of 150×150 pixels, while 316×252 pixels for the last one.

6.3 Evaluating the estimator

To evaluate the correctness of the motion estimator, we calculate error between estimated and real optical flows using the equation below:

$$\alpha_e = \arccos\left(\frac{uu_r + vv_r + 1}{\sqrt{u^2+v^2+1}\sqrt{u_r^2+v_r^2+1}}\right)$$

where a_e is the angular error for a given pixel with (u,v) the estimated and (u_r,v_r) the real motion vectors. We used "treetran" and "treediv" image sequences for the evaluation, showing translational and divergent motion respectively [3]. The results obtained using "treetran" and "treediv" image sequences are shown in table 2.

Table 2 Evaluating the M-estimator

	Angular error			
	treetran		treediv	
	$\bar{x}$	σ	$\bar{x}$	σ
Matlab	1.63	5.27	6.06	8.22
C	1.10	0.99	4.15	2.69
CUDA	1.19	1.00	5.73	3.91

6.4 Precision

Most of the complex scientific applications developed for high performance computing desire more precision to get more accurate results. But, the GPUs are specialized to perform many single-precision floating-point operations, though newer cards like NVIDIA Geforce GTX 480 consists of 480 cores capable of both single and double-precision operations.

The vision algorithm implemented in CUDA is ported from MATLAB code, where all the computations are done entirely in double-precision; fortunately, the effects of low-precision in parallel implementation are not obvious. The main reason is the type of algorithm whether it can produce acceptable results, or ones that are usable. Here the resulting saliency map may be inaccurate, but visually fine with universal image quality index [19] of 99.66% and 2-digit precision among the 24-bits of float mantissa. The figure 8 shows the mean error with respect to the reference during different stages of the pathway. We observe that the accuracy of the results increases along the progressing stages because of the reduction of information, more evident during Gabor filtering and normalization phases until finally ending up in regions that are salient.

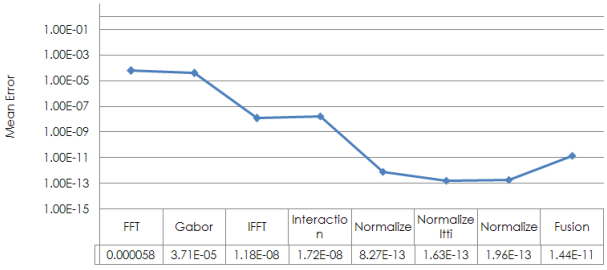


Fig. 8 The effect of lower precision support on the result

6.5 Other Optimizations

In our implementation, the use of constant memory cache shows almost negligible effect on overall performance. With access speed similar to registers, it can help to save registers that can be used for other optimizations. Here, tex-

Table 3 Speedups after optimizations to static pathway

Case	Over C	Over First CUDA	MPixels/sec
First Implementation	396x	1.00x	4.11
Textures used	421x	1.06x	5.93
No bank conflicts	429x	1.08x	6.08
Fast math used	440x	1.11x	6.25

Table 4 Computational cost of each step in static pathway

Kernel	Geforce GTX 480 (ms)
Retina	9.74
Mask	0.12
FFT	0.45
Shift	0.09
24×Gabor	1.58
24×Inverse shift	0.99
24×IFFT	10.85
24×Interaction	3.45
24×Normalize	3.08
24×Normalize Itti	3.10
24×Normalize Fusion	2.66
Memory transfers	7.64
Total	43.75

ture cache is used to store predefined readonly masks, resulting in a speedup of 1.06x over first CUDA implementation and 421x over CPU implementation. Also, the use of shared memory without bank conflicts resulted in speedup of 429x over C and 1.08x over first CUDA implementation. As a last optimization is the use of compiler option -use_fast_math to CUDA math library instead of standard math library. The option worked after few tweaks, and resulted in 440x speedup over C implementation and 1.11x over first CUDA implementation. The results of the different optimizations are presented in table 3.

6.6 Using different graphics cards

The CUDA implementation is tested against on the newest Geforce GTX 480. The device has 15 streaming multiprocessor with total 480 cores of clock rates 1.48 GHz each, providing 1.35 TFLOPS of single-precision and 168 GFLOPS of double precision computational power with memory bandwidth 177.4 GB/sec.

In tables 4 and 5, the computational cost for each step of static and dynamic pathways of the visual saliency model with image sizes of 512×512 and 256×256 pixels respectively. Furthermore, the figure 9 presents a plot of execution times for different image sizes on 8800 GTS and GTX 260, and clearly the execution speedups with increasing image sizes on the faster Geforce GTX 260 because of more execution units to use.

Table 5 Computational cost of each step in dynamic pathway

Kernel	Geforce GTX 480 (ms)
MCPI	22.68
Ver. Gaussian recursive	21.90
Hor. Gaussian recursive	11.06
Demodulation	1.4
Modulation	0.39
Retinal Filtering	9.74
Gradients	1.17
Misc.	0,25
Memory transfers	0,56
Total	69.16

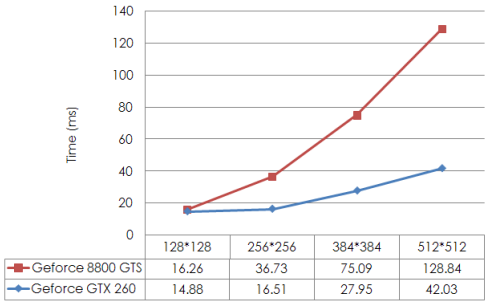


Fig. 9 Timings for different image sizes from different cards

7 Conclusion

In the article, we have implemented in real-time a very recent visual saliency model that is based on the human visual system. This real-time processing for such models would create an opportunity of inclusion of many other complex processes or pathways into the existing model, for example: color, face recognition, audio, and many more. Also, the real-time capability enables it to be used as a powerful tool in many other applications like robotics vision.

The CUDA programming model for general-purpose computations on graphics device is well-suited for almost all algorithms that exhibit data-parallelism. The new model launches a large number of threads, and does very fast context switching among them to hide the memory latency. Also, this is facilitated by the inclusion of new hardware shared memory that reduces the number of memory accesses to the global memory. The motivation of this project is to implement such algorithms on modern graphic devices to use their raw computational power; ending up with enormous speedups. Our implementation of visual saliency model confirmed that the algorithm effectively maps onto the graphics architecture. Thus, efficiently utilizing the highly data-parallel processing capabilities of the graphics hardware; resulting in tremendous speedups of more than 440x. Despite, using reduced precision that somewhat affected the overall accuracy during different phases of the algorithm but the final result is adequate, and visually these differences are undetectable.

Moreover, the implementation is demonstrated using a typical webcam interfaced using OpenVIDIA [6] library on an ordinary computer system. Real-time processing is achieved for an image of size 320×240 pixels running at 22fps.

Discussion: The programming model for graphics devices is scalable, and can be easily upgraded rather than when using specialized hardware, SIMD systems, or supercomputers. According to the Moore's law, the peak performance of GPUs is increasing by a factor of two and a half times per year. The new Tesla 20 series devices are targeting a double-precision peak performance ranging from 520-620 GFLOPS , and more interestingly single-precision floating-point computations will top into TFLOPS. This development alongside a flexible programming model will interest the GPGPU community.

A number of attempts have been made to design a specialized system using multiple GPUs as clusters, where multiple vision algorithms are computed on separate cards or work is assigned explicitly to each. Although, parallel GPU processing is difficult to implement using current CUDA programming model, where multiple graphics devices can not share data and the overhead of copying back partial results to host CPU is very high. The newer versions of the programming models are expected to handle multi-GPUs more easily, and will open a new era in high performance computing. Consequently, GPUs will find their application in areas like biological engineering, oil and gas exploration, and financial analysis. In future, the introduction of devices with more execution units accompanied by more flexible programming model will help to put away the doubts in GPGPU community.

References

1. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide (2007)
2. Allusse, Y., Horain, P., Agarwal, A., Saipriyadarshan, C.: Gpucv: an opensource gpu-accelerated framework forimage processing and computer vision. In: MM '08: Proceeding of the 16th ACM international conference on Multimedia, pp. 1089–1092. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1459359.1459578
3. Bruno, E., Pellerin, D.: Robust motion estimation using spatial gabor-like filters. Signal Process **82**, 297–309 (2002)
4. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: stream computing on graphics hardware. ACM Trans. Graph. **23**, 777–786 (2004)
5. Chalmers, A., Debattista, K., Sundstedt, V., Longhurst, P., Gillibrand, R.: Rendering on demand. In: Eurographics Symposium on Parallel Graphics and Visualization (2006)
6. Fung, J., Mann, S., Aimone, C.: Openvidia: Parallel gpu computer vision. In: MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia (2005)
7. Group, K.: Opencl - the open standard for parallel programming of heterogeneous systems. URL `http://www.khronos.org/opencl/`
8. Itti, L.: Real-time high-performance attention focusing in outdoors color video streams. In: Rogowitz, B., Pappas, T.N. (eds.) Proc. SPIE Human Vision and Electronic Imaging VII (HVEI'02), San Jose, CA, pp. 235–243. SPIE Press (2002)
9. Itti, L., Koch, C., Niebur, E.: A model of saliency-based visual attention for rapid scene analysis. IEEE Trans. Pattern Anal. Mach. Intell. **20**, 1254–1259 (1998)
10. Lee, S., Kim, G.J., Choi, S.: Real-time tracking of visually attended objects in interactive virtual environments. In: VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology (2007)
11. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: A unified graphics and computing architecture. IEEE Micro **28**, 39–55 (2008)
12. Longhurst, P., Debattista, K., Chalmers, A.: A gpu based saliency map for high-fidelity selective rendering. In: AFRIGRAPH 2006 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (2006)
13. Mantiuk, R., Myszkowski, K., Pattanaik, S.: Attention guided mpeg compression for computer animations. In: SCCG '03: Proceedings of the 19th spring conference on Computer graphics (2003)
14. Marat, S., Ho Phuoc, T., Granjon, L., Guyader, N., Pellerin, D., Guérin-Dugué, A.: Modelling spatio-temporal saliency to predict gaze direction for short videos. Int. J. Comput. Vision **82**, 231–243 (2009)
15. Odobez, J.M., Bouthemy, P.: Robust multiresolution estimation of parametric motion models applied to complex scenes. Journal of Visual Communication and Image Representation **6**, 348–365 (1994)
16. Ouerhani, N., Hgli, H.: Real-time visual attention on a massively parallel simd architecture. Real-Time Imaging **9**, 189–196 (2003)
17. Peters, C.: Toward 3D selection and skeleton construction by sketching. In: Eurographics Ireland 2007 (2007)
18. Pichon, E., Itti, L.: Real-time high-performance attention focusing for outdoors mobile beobots. In: Proc. AAAI Spring Symposium, Stanford, CA (AAAI-TR-SS-02-04) (2002)
19. Wang, Z., Bovik, A.C.: A universal image quality index. Signal Processing Letters, IEEE **9**, 81–84 (2002)
20. Xu, T., Muhlbauer, Q., Sosnowski, S., Kühnlenz, K., Buss, M.: Looking at the surprise: Bottom-up attentional control of an active camera system. In: ICARCV (2008)